DAM
Desarrollo de Aplicaciones Multiplataforma
2º Curso

AD
Acceso a Datos

UD 3
Manejo de Conectores

IES BALMIS
Dpto Informática
Curso 2019-2020
Versión 1 (03/2019)

UD3 – Manejo de Conectores

ÍNDICE

1. Introducción

Actualmente, las **bases de datos relacionales** constituyen el sistema de almacenamiento probablemente más extendido, aunque otros sistemas de almacenamiento de la información se estén abriendo paso poco a poco.

Una **base de datos relacional** se puede definir, de una manera simple, como aquella que presenta la información en tablas con filas y columnas.

Una **tabla o relación** es una colección de objetos del mismo tipo (filas o tuplas).

En cada tabla de una base de datos se elige una **clave primaria** para identificar de manera unívoca a cada fila de la misma.

El **Sistema Gestor de Bases de Datos**, en inglés conocido como **Database Management System (DBMS)** gestiona el modo en que los datos se almacenan, mantienen y recuperan.

En el caso de una base de datos relacional, el sistema gestor de base de datos se denomina **Relational Database Management System (RDBMS)**.

Tradicionalmente, la programación de bases de datos ha sido como una torre de Babel: gran cantidad de productos de bases de datos en el mercado y cada uno "hablando" en su lenguaje privado con las aplicaciones.

Java, mediante **JDBC (Java Database Connectivity)**, permite simplificar el acceso a bases de datos relacionales, proporcionando un lenguaje mediante el cual las aplicaciones pueden comunicarse con motores de bases de datos. Sun desarrolló este API para el acceso a bases de datos, con tres objetivos principales en mente:

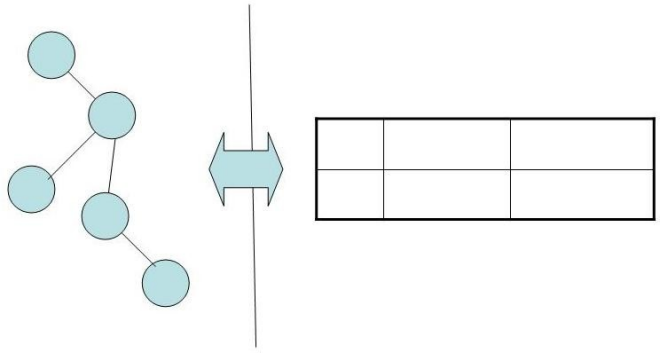
- Ser un **API con soporte de SQL**: poder construir sentencias SQL e insertarlas dentro de llamadas al API de Java.
- **Aprovechar** la experiencia de los API's de bases de datos existentes.
- Ser lo más **sencillo** posible.

1.1 El desfase objeto-relacional

El desfase objeto-relacional, también conocido como impedancia objeto-relacional, consiste en la diferencia de aspectos que existen entre la programación orientada a objetos y la base de datos. Estos aspectos se puede presentar en cuestiones como:

- **Lenguaje de programación**: el programador debe conocer el lenguaje de programación orientado a objetos (POO) y el lenguaje de acceso a datos.
- **Tipos de datos**: en las bases de datos relacionales siempre hay restricciones en el uso de tipos, mientras que la programación orientada a objetos utiliza tipos de datos mas complejos.
- **Paradigma de programación**: en el proceso de diseño y construcción del software se tiene que hacer una traducción del modelo orientado a objetos de clases al modelo Entidad-Relación (E/R) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas (o filas), lo que implica que se tengan que diseñar dos diagramas diferentes para el diseño de la aplicación.

El modelo relacional trata con relaciones y conjuntos debido a su naturaleza matemática. Sin embargo, el modelo de **POO** trata con objetos y las asociaciones entre ellos. Por esta razón, el problema entre estos dos modelos surge en el momento de querer persistir los objetos de negocio.



La escritura (y de manera similar la lectura) mediante JDBC implica:

- Abrir una conexión.
- Crear una sentencia en SQL.
- Copiar todos los valores de las propiedades de un objeto en la sentencia, ejecutarla y así almacenar el objeto.

Esto es sencillo para un caso simple, pero complicado si el objeto posee muchas propiedades, o bien se necesita almacenar un objeto que a su vez posee una colección de otros elementos. Se necesita crear mucho más código, además del tedioso trabajo de creación de sentencias SQL.

Este problema es lo que denominábamos **impedancia Objeto-Relacional**, o sea, el conjunto de dificultades técnicas que surgen cuando una base de datos relacional se usa en conjunto con un programa escrito en POO.

Como ejemplo de desfase objeto-relacional, podemos poner el siguiente: supón que tienes un objeto "Agenda Personal" con un atributo que sea una colección de objetos de la clase "Persona". Cada persona tiene un atributo "teléfono". Al transformar este caso a relacional, se ocuparía más de una tabla para almacenar la información, conllevando varias sentencias SQL y bastante código.

2. Protocolos de acceso a bases de datos

Hace años, cuando Sun Microsystems desarrolló Java, uno de los aspectos que tuvieron que pensar fue la manera de enfocar el acceso a datos, todo lo concerniente a los protocolos de acceso a bases de datos.

Debido a la confusión que había por la proliferación de API's propietarios de acceso a datos, Sun buscó los aspectos de éxito de un API de este tipo como el **ODBC (Open Database Connectivity)**. ODBC había sido desarrollado por Microsoft con la idea de tener un estándar para el acceso a bases de datos en entorno Windows.

Aunque la industria aceptó ODBC como medio principal para acceso a bases de datos en Windows, la verdad es que no se introduce bien en el mundo Java, debido a la complejidad que presenta ODBC, y que entre otras cosas ha impedido su transición fuera del entorno Windows.

ODBC (Open Database Connectivity)
https://es.wikipedia.org/wiki/Open_Database_Connectivity

La idea en el desarrollo de JDBC era intentar ser tan sencillo como fuera posible, pero proporcionando a los desarrolladores la máxima flexibilidad.

Una primera clasificación podría ser los que funcionan abriendo ficheros de datos directamente o los que disponen de una arquitectura cliente/servidor:

SGBD con acceso directamente a ficheros

Microsoft Access, integrada en Microsoft Office y muy usada en aplicaciones de uso doméstico.

SQLite se emplea mucho para bases de datos de pequeño tamaño incrustadas en aplicaciones. Muy usada en entornos de dispositivos móviles. Le falta alguna funcionalidad al acceder desde Java, como por ejemplo, cursor SCROLLABLE al realizar un SELECT lo que permite moverse hacia delante y atrás.

Apache Derby, aunque también tiene la posibilidad de trabajar en cliente/servidor, se suele usar en pequeñas aplicaciones pero con datos más complejos.

Arquitectura Cliente/Servidor

En la arquitectura denominada **Cliente/Servidor** podemos distinguir entre máquina cliente, donde trabaja el usuario que se conecta a la base de datos, y máquina servidor, aquella en la que se aloja y ejecuta el sistema de bases de datos.

Algunos ejemplos de SGBD que pueden trabajar con arquitectura cliente/servidor son:

- **Oracle**, posiblemente el más empleado a nivel de empresa.
- **MySQL** y **PostgreSQL**, muy utilizados en entorno de servidores de Intenet.
- **Microsoft SQL Server**, utilizado en PYMES con servidores de Microsoft.
- **Firebird**, utilizado en entornos PYME.

Uno de los aspectos más importantes en esta arquitectura es el puerto por el que escucha cada SGBD, ya que en cada cliente deberemos configurar al menos:

- Tipo de SGBD (para establecer el protocolo de SQL)
- IP de la máquina servidor
- Puerto de escucha
- Usuario
- Contraseña

El puerto por defecto donde escucha cada SGBD es diferente y está consensuado:

SGBD	*Puerto*
Oracle	1521
MySQL	3306
SQL Server (Microsoft)	1433
Postgres	5432
Firebird	3050
DB2 (IBM)	50000

En el caso de otros SGBD no relacionales los puertos son:

SGBD	*Puerto*
eXist-db	8080
MongoDB	27017

En muchas ocasiones, las instalaciones proporcionan otros puertos para realizar la administración, instalar servicios web, acceder con cifrado, etc. Por ejemplo:

- **Oracle** instala en el **5500** el servicio web Enterprise Manager para la administración y monitorización
- **eXist-db** instala en el **8443** un servicio de acceso cifrado

Otros servicios pueden ocupar estos puertos, como los de **proxy** que suelen instalarse en algunos casos en el **8080.**

Para comprobar el estado de los puertos de escucha, y en caso de estar ocupados utilizar otros, podemos usar los siguientes comandos del sistema operativo:

Windows	`netstat -a -p TCP | find "LISTENING"`
Linux	`netstat -at`

Formas de conexión

Existen distintas formas de conectar a gestores de bases de datos, como por ejemplo:

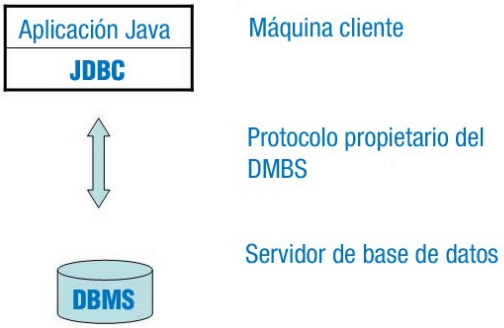
- Bases de datos incrustadas ("**embedded**"), en las que la conexión entre el programa y la base de datos es directa. Puede permitir la mayor velocidad en la comunicación, pero supone hacer cambios (quizá profundos) en el programa en caso de necesitar conectar a una base de datos distinta, lo que no es deseable.
- Como alternativa, es preferible usar un API (interfaz de programación de aplicaciones) genérico, que permita conectar de la misma forma a distintos gestores de bases de datos, así como enviar peticiones y recibir los datos de registros con las respuestas de una manera uniforme. Uno de los estándares más extendidos es **ODBC (Open DataBase Connectivity)**, usado habitualmente en C, C++, C# y Visual Basic.
- ODBC es poco adecuado para ser usado desde Java, porque se basa en llamadas en lenguaje C (lo que implica uso de punteros, que no existen en Java), y la conversión puede suponer riesgos de seguridad y de falta robustez. Por eso, en Java se suele emplear una alternativa totalmente basada en objetos, llamada **JDBC (Java Database Connectivity)**.

En la mayoría de casos, se deberá instalar un "**driver**" que permita acceder desde el lenguaje o herramienta que se está utilizando al gestor de bases de datos deseados. En el mundo Java es habitual llamar también **conector** a estos drivers.

2.1 Arquitectura JDBC

El API JDBC soporta dos modelos de procesamiento para acceso a bases de datos: de dos y tres capas.

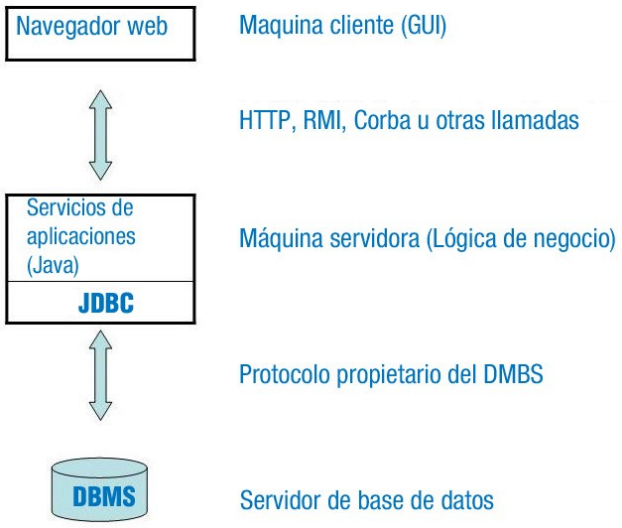
En el **modelo de dos capas**, una aplicación se comunica directamente a la fuente de datos. Esto necesita un conector JDBC que pueda comunicar con la fuente de datos específica a la que acceder.



Los comandos o instrucciones del usuario se envían a la base de datos y los resultados se devuelven al usuario. La fuente de datos puede estar ubicada en otra máquina a la que el usuario se conecte por red. A esto se denomina configuración cliente/servidor, con la máquina del usuario como cliente y la máquina que aloja los datos como servidor.

En el **modelo de tres capas**, los comandos se envían a una capa intermedia de servicios, la cual envía los comandos a la fuente de datos.

La fuente de datos procesa los comandos y envía los resultados de vuelta la capa intermedia, desde la que luego se le envían al usuario.



El **API JDBC** viene distribuido en dos paquetes:

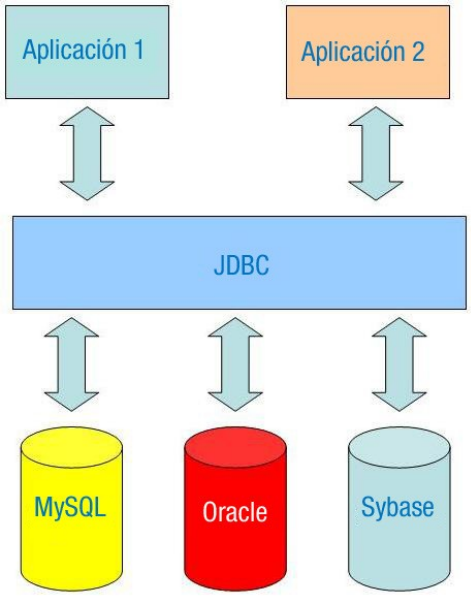
- **java.sql**, dentro de J2SE.
- **javax.sql**, extensión dentro de J2EE.

JDBC
https://www.adictosaltrabajo.com/2006/05/04/introjdbc/

2.2 Conectores o Drivers

Un **conector o driver** es un conjunto de clases encargadas de implementar los interfaces del API y acceder a la base de datos.

Para poder conectarse a una base de datos y lanzar consultas, una aplicación necesita tener un driver adecuado. Un conector suele ser un fichero .jar que contiene una implementación de todas las interfaces del API JDBC.



Cuando se construye una aplicación de base de datos, JDBC oculta lo específico de cada base de datos, de modo que el programador se ocupe sólo de su aplicación.

El conector lo proporciona el fabricante de la base de datos o bien un tercero.

El código de nuestra aplicación no depende del driver, puesto que trabajamos mediante los paquetes java.sql y javax.sql.

JDBC ofrece las clases e interfaces para:

- Establecer una conexión a una base de datos.
- Ejecutar una consulta.
- Procesar los resultados.
- Cerrar recursos

Ejemplo:

```
// Establece la conexión
Connection con = DriverManager.getConnection("jdbc:driver:URL",
                    "miLogin", "miPassword" );

// Ejecuta y procesa la consulta
Statement stmt = (Statement) con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT nombre, edad FROM Empleados");
while (rs.next()) {
      String nombre = rs.getString("nombre");
      int edad = rs.getInt("edad");
}
rs.close();

// Cerrar la conexión
con.close();
```

En principio, todos los conectores deben ser compatibles con ANSI SQL-2 Entry Level (ANSI SQL-2 se refiere a los estándares adoptados por el American National Standards Institute en 1992. Entry Level se refiere a una lista específica de capacidades de SQL.) Los desarrolladores de drivers pueden establecer que sus conectores conocen estos estándares.

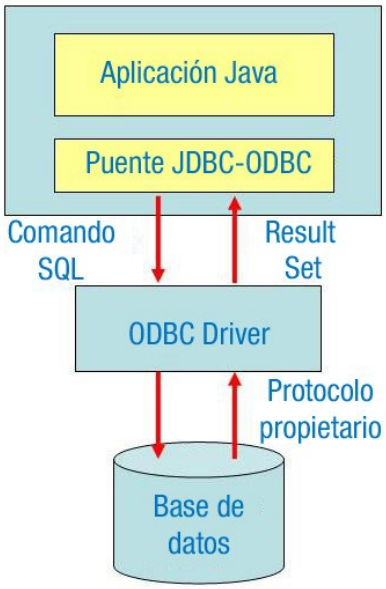
Hay cuatro tipos de drivers JDBC: Tipo 1, Tipo 2, Tipo 3 y Tipo 4, que veremos a continuación.

JDBC y otras tecnologías de Oracle
https://www.oracle.com/technetwork/java/javase/jdbc/index.html

2.3 Conectores tipo 1, tipo 2, tipo3 y tipo 4

Los conectores tipo 1 se denominan también JDBC-ODBC Bridge (puente JDBC-ODBC).

Proporcionan un puente entre el API JDBC y el API ODBC. El driver **JDBC-ODBC Bridge** traduce las llamadas JDBC a llamadas ODBC y las envía a la fuente de datos ODBC.



Como ventajas destacar:

- No se necesita un driver específico de cada base de datos de tipo ODBC.
- Está soportado por muchos fabricantes, por lo que tenemos acceso a muchas Bases de Datos.

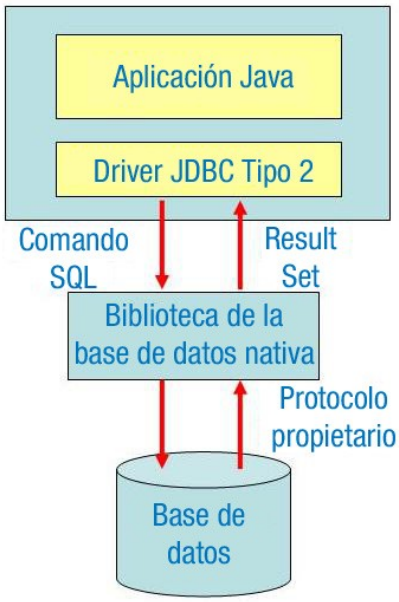
Como desventajas señalar:

- Hay plataformas que no lo tienen implementado.
- El rendimiento no es óptimo ya que la llamada JDBC se realiza a través del puente hasta el conector ODBC y de ahí al interface de conectividad de la base de datos. El resultado recorre el camino inverso.
- Se tiene que registrar manualmente en el gestor de ODBC teniendo que configurar el DSN (Data Source Names, Nombres de fuentes de datos).

Este tipo de driver está incluido en el JDK hasta la versión 7.
En Java 8, el JDBC-ODBC Bridge ha sido elimnado.

Los conectores tipo 2 se conocen también como: API nativa

Convierten las llamadas JDBC a llamadas específicas de la base de datos para bases de datos como SQL Server, Informix, Oracle, o Sybase.



El conector tipo 2 se comunica directamente con el servidor de bases de datos, por lo que es necesario que haya código en la máquina cliente.

Como ventaja, este conector destaca por ofrecer un rendimiento notablemente mejor que el JDBC-ODBC Bridge.

Como inconveniente, señalar que la librería de la bases de datos del vendedor necesita cargarse en cada máquina cliente. Por esta razón los drivers tipo 2 no pueden usarse para Internet.

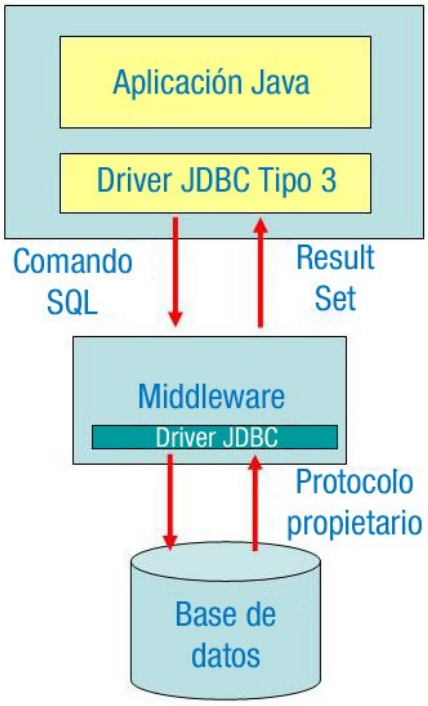
Los drivers Tipo 1 y 2 utilizan código nativo vía JNI, por lo que son más eficientes.

JNI – Java Native Interface
https://es.wikipedia.org/wiki/Java_Native_Interface

DSN – Data Source Name
https://en.wikipedia.org/wiki/Data_source_name

Tipo 3: JDBC-Net pure Java driver.

Tiene una aproximación de tres capas. Las peticiones JDBC a la base de datos se pasan a través de la red al servidor de la capa intermedia (middleware). Este servidor traduce este protocolo independiente del sistema gestor a protocolo específico del sistema gestor y se envía a la base de datos. Los resultados se mandan de vuelta al middleware y se enrutan al cliente.



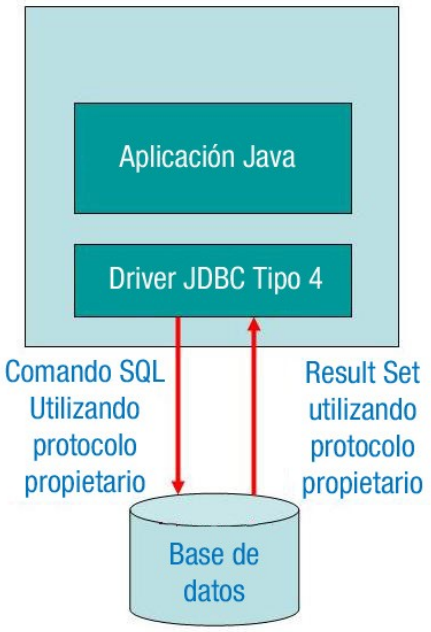
Es útil para aplicaciones en Internet.

Este driver está basado en servidor, por lo que no se necesita ninguna librería de base de datos en las máquinas clientes.

Normalmente, un driver de tipo 3 proporciona soporte para balanceo de carga, funciones avanzadas de administrador de sistemas tales como auditoría, etc.

Tipo 4: Protocolo nativo.

En este caso se trata de conectores que convierten directamente las llamadas JDBC al protocolo de red usando por el sistema gestor de la base de datos. Esto permite una llamada directa desde la máquina cliente al servidor del sistema gestor de base de datos y es una solución excelente para acceso en intranets.



Como ventaja se tiene que no es necesaria traducción adicional o capa middleware, lo que mejora el rendimiento, siendo éste mejor que en el caso de los tipos 1 y 2.

Además, no se necesita instalar ningún software especial en el cliente o en el servidor.

Como inconveniente, de este tipo de conectores, el usuario necesita un driver diferente para cada base de datos.

Un ejemplo de este tipo de conector es **Oracle Thin**.

3. Conexión a una base de datos

Para acceder a una base de datos y así poder operar con ella, lo primero que hay que hacer es conectarse a dicha base de datos.

En Java, para establecer una conexión con una base de datos podemos utilizar el método **getConnection**() de la clase **DriverManager**. Este método recibe como parámetro la URL de JDBC que identifica a la base de datos con la que queremos realizar la conexión.

La ejecución de este método devuelve un objeto Connection que representa la conexión con la base de datos.

Cuando se presenta con una URL específica, DriverManager itera sobre la colección de drivers registrados hasta que uno de ellos reconoce la URL especificada. Si no se encuentra ningún conector adecuado, se lanza una **SQLException**

MySQL driver JDBC - Download
https://dev.mysql.com/downloads/connector/j/5.1.html

Veamos un ejemplo primer ejemplo con comentarios para conectarnos a una base de datos MySQL

URL de conexión con MySQL

Una URL tiene dos partes:

Principal	protocolo://ip:puerto/recurso
Separación	?
Parámetros	param1=valor&param2=valor&...

En nuestro caso:

jdbc	Establece que es una URL de driver jdbc
:mysql	Indica que la conexión es a MySQL
://localhost	IP del servidor
:3306	Puerto del servicio de MySQL
/world	BD a la que conectamos
?user=root	Usuario con el conectamos
&password=1234	Contraseña del usuario
&useSSL=false	Indica si la conexión usará cifrado SSL o no. En caso de no existir SSL en el servidor, es obligatorio poener false.
&autoReconnect=true	Si la conexión se rompe, con el valor a true intenta de nuevo conectar, mientras que con el valor a false devuelve "exception"
&zeroDateTimeBehavior=convertToNull	Establece qué hacer cuando el driver encuentra un valor DATETIME compueto completamente por ceros, es decir, '0000-00-00 00:00:00' Las posibilidades son: • Dar una "exception", • Devolver un null "convertToNull".

Existen valores por defecto para algunos aspectos en caso de no estar presentes en la URL:

:3306	Si no se indica, el valor por defecto es 3306
/world	La BD a la que conectamos no es obligatoria. Si no la indicamos estaremos conectados a MySQL pero no existirá una BD seleccionada.
&useSSL=true	Por defecto es **true** para la mayoría de las versiones MySQL 5.X y 8.X
&autoReconnect=false	Por defecto es **false**
&zeroDateTimeBehavior=exception	Por defecto genera una **exception**

Más parámetros de la URL de conexión utilizando JDBC
https://dev.mysql.com/doc/connector-j/5.1/en/connector-j-reference-configuration-properties.html

Vemos un ejemplo de conexión a MySQL:

MySQL01.java

```
public static void main(String[] args) {
  try {
     // Cargar el driver de mysql
     Class.forName("com.mysql.jdbc.Driver");

     // Cadena de conexión a MySQL
     String conURL = "jdbc:mysql://localhost?user=root&password=1234&useSSL=false";

     // Obtener la conexión
     Connection con = DriverManager.getConnection(conURL);

     // Consultar la versión de MySQL
     Statement stmt = con.createStatement();
     ResultSet rs = stmt.executeQuery("SELECT version() as versionmysql");
     if (rs.next()) {
        System.out.println("VERSIÓN DE MYSQL = " + rs.getString("versionmysql"));
     } else {
        System.out.println("Consulta sin resultados");
     }

     // Cerrar la conexión
     con.close();

  } catch (SQLException e) {
     System.out.println("SQL Exception: " + e.toString());
  } catch (ClassNotFoundException cE) {
     System.out.println("Exception: " + cE.toString());
  }
}
```

En el proyecto deberemos añadir la librería **jar** de **driver mysql**.
Todos los import serán de la librería **java.sql.***

4. Operaciones: ejecución de consultas

Para operar con una base de datos, ejecutando las consultas necesarias, nuestra aplicación deberá hacer:

- **Cargar** el driver necesario para comprender el protocolo que usa la base de datos en cuestión.
- **Establecer** una conexión con la base de datos.
- **Enviar** consultas SQL y procesar el resultado.
- **Liberar** los recursos al terminar.
- **Gestionar** los errores que se puedan producir.

Podemos utilizar los siguientes tipos de sentencias:

- **Statement**: para sentencias sencillas en SQL.
- **PreparedStatement**: para consultas preparadas, como por ejemplo las que tienen parámetros.
- **CallableStatement**: para ejecutar procedimientos almacenados en la base de datos.

El API JDBC distingue dos tipos de consultas:

- Consultas: SELECT
- Actualizaciones: INSERT, UPDATE, DELETE, sentencias DDL.

4.1 Ejemplo: consultas con MS-Access

Veamos a continuación cómo realizar una consulta en la base de datos que creamos en el apartado anterior, la base de datos **BD-Farmacia.mdb**.

En este caso utilizaremos un acceso mediante el driver JDBC **UcanAccess**. UCanAccess requiere Java 6 o posterior para ejecutarse.

Añadiremos todos las librerías jar del driver al proyecto.

Dentro del código se carga el driver antes de acceder a la base de datos:

```
Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");
```

Posteriormente, una vez instalado el conector y cargado dentro del código Java sólo necesitamos **realizar la conexión** a la base de datos para comenzar a trabajar con ella. La clase **DriverManager** define el método **getConnection** para crear una

conexión a una base de datos. Este método toma como parámetro una URL JDBC donde se indica el sistema gestor y la base de datos. Opcionalmente, y dependiendo del sistema gestor, habrá que especificar el login y password para la conexión. En el caso de JDBC la cadena de conexión sería:

jdbc:ucanaccess://path/archivo.mdb

Cuando se trata de bases de datos grandes y se usa la configuración predeterminada de "memoria" (es decir, con la propiedad de controlador memory=true), se recomienda que los usuarios asignen suficiente memoria a la JVM usando las opciones -Xms y -Xmx. De lo contrario, será necesario establecer la propiedad "memoria" del controlador en "falso" (memory=false):

Finalmente, el código para establecer una conexión quedaría del siguiente modo:

```
String connectionUrl="jdbc:ucanaccess://./datos/BD-Farmacia.mdb;memory=false";
Connection con = DriverManager.getConnection(connectionUrl);
```

Las consultas que se realizan a una base de datos se realizan utilizando objetos de las clases **Statement** y **PreparedStatement**. Estos objetos se crean a partir de una conexión.

```
Statement stmt = con.createStatement();
```

La clase Statement contiene los métodos **executeQuery** y **executeUpdate** para realizar consultas y actualizaciones, respectivamente. Ambos métodos soportan consultas en SQL-92. Así por ejemplo, para obtener los nombres de los medicamentos que tenemos en la tabla medicamentos, de la base de datos farmacia.mdb que creamos anteriormente, tendríamos que emplear la sentencia:

```
ResultSet rs = stmt.executeQuery("SELECT nombre from medicamentos");
```

El método **executeQuery** devuelve un objeto **ResultSet** para poder recorrer el resultado de la consulta utilizando un cursor.

```
while (rs.next()) {
      String usuario = rs.getString("nombre");
      ...
```

El método **next** se emplea para hacer avanzar el cursor. Para obtener una columna del registro utilizamos los métodos **get**. Hay un método get para cada tipo básico Java y para las cadenas.

Comentar que un método interesante del cursor es **wasNull** que nos informa si el último valor leído con un método **get** es nulo.

Respecto a las consultas de actualización, **executeUpdate**, retornan el número de registros insertados, registros actualizados o eliminados, dependiendo del tipo de consulta que se trate.

Aquí tienes el proyecto que realiza la consulta de todos los medicamentos de la tabla que los contiene, llamada medicamentos, en la base de datos **BD-Farmacia.mdb**.

BDAccess01.java

```
public static void main(String[] args) {
  try {

    // Cargar el driver de Microsoft Access
    Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");
    String connectionUrl = "jdbc:ucanaccess://./datos/BD-Farmacia.mdb;memory=false";

    // Obtener la conexión
    Connection con = DriverManager.getConnection(connectionUrl);

    // La clase Statement contiene los métodos executeQuery y
    //      executeUpdate para realizar consultas y actualizaciones
    Statement stmt = con.createStatement();

    // El método executeQuery devuelve un objeto ResultSet para poder
    // recorrer el resultado de la consulta utilizando un cursor.
    // Esta consulta obtiene todos los datos, todos los campos,
    // (debido al SELECT *), almacenados en la tabla medicamentos.
    ResultSet rs = stmt.executeQuery("SELECT * from medicamentos");

    // Mientras queden datos
    while (rs.next()) {
      // Imprimir en la consola
      String codigo = rs.getString("codigo");
      String nombre = rs.getString("nombre");
      String precio = rs.getString("precio");
      String pvp    = rs.getString("pvp");
      String unidades = rs.getString("unidades");
      System.out.println(codigo+"---"+nombre+"--"+precio+"--"+pvp+"--"+unidades);
    }
    rs.close();

    con.close();

  } catch (SQLException e) {
    System.out.println("SQL Exception: "+ e.toString());
  } catch (ClassNotFoundException cE) {
    System.out.println("Excepción: "+ cE.toString());
  }
}
```

En el proyecto deberemos añadir la librería **jar** de **driver ucanaccess**.
Todos los import serán de la librería **java.sql.***

UCanAccess driver JDBC - Download
https://sourceforge.net/projects/ucanaccess/

El driver UCanAccess está compuesto por 5 archivos jar.